Chapter 6 Task Structuring

Task structuring aims to allocate transformations from a data/control flow diagram to tasks in a concurrent design. This goal is accomplished by applying knowledge about how to structure tasks. Task Structuring Knowledge is organized as five, distinct, decision-making processes, shown in Figure 25. The main information used to structure tasks consists of a fully-classified data/control flow diagram, as described in Chapter 4, and the state of the evolving, concurrent design. The main information created by applying Task Structuring Knowledge consists of a set of tasks that become components in a concurrent design.

Task Structuring Knowledge embodies a relatively straightforward decision-making strategy. First, the transformations represented in the data/control flow diagram are examined to identify those that might form the basis for a task. This process is facilitated by the concept classification performed earlier during the analysis of the input specification (refer to Chapter 4, section 4.3). Once the initial set of tasks is determined, all remaining, unallocated, transformations (that is, those that do not provide a basis for forming a task) from the data/control flow diagram are allocated among the set of tasks. Once all transformations are allocated to tasks, a number of criteria for combining tasks are applied; thus, the initial set of candidate tasks might possibly be reduced. After tasks are combined, a final look at the task structure determines whether

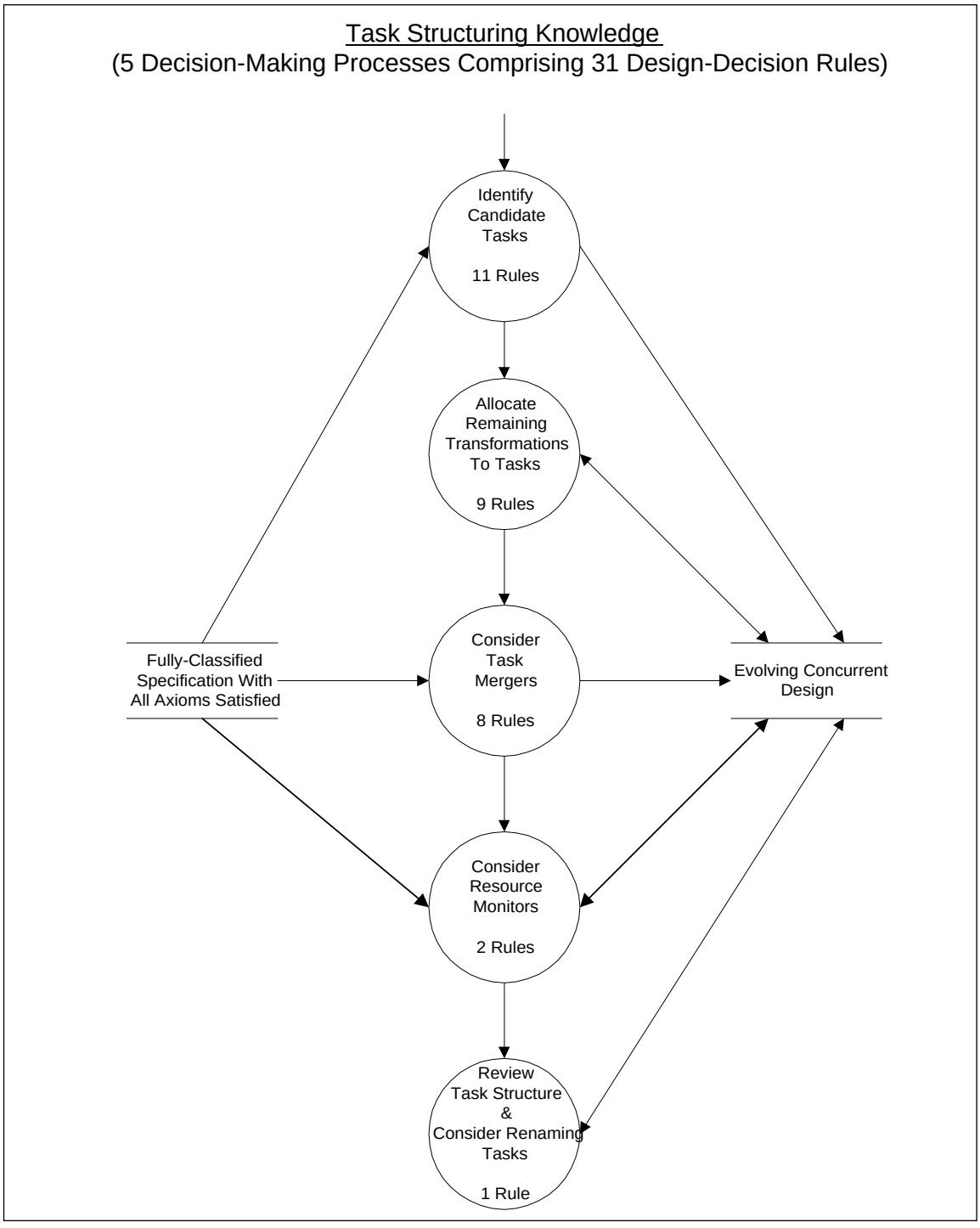

Figure 25. Organization of Task Structuring Knowledge

any tasks are needed to monitor shared resources. Finally, the designer is offered an opportunity to review the task structure, and to consider whether any of the tasks should be assigned more appropriate names.[1] To implement the task-structuring strategy, each of the decision-making processes, shown in Figure 25, comprises a set of design-decision rules that are based on task-structuring heuristics included within the CODARTS design method. [Gomaa93, Chapter 14]

6.1 Identify Candidate Tasks

Task structuring begins with a decision-making process that: 1) identifies those transformations in the input specification that can be allocated to a task, 2) makes the necessary allocations, 3) captures the decisions and rationale, and 4) denotes the traceability between existing specification elements and newly created design elements. The decision-making process consists of a set of rules that look for input/output tasks and for internal tasks based on the presence of certain types of transformations within the input specification. For each appropriate transformation found, one task is added to the evolving, concurrent design. In the following subsections, the rules for identifying input/output tasks are defined first, followed by the rules for identifying internal tasks.

6.1.1 Rules for Identifying Input/Output Tasks

According to CODARTS, each asynchronous device, periodic device, or external subsystem represented on the context diagram of an input specification must be managed

[1] To facilitate automated generation of designs, the design-decision rules use algorithms for constructing the names of design elements. Often these algorithms create names, readily traceable to elements from the input specification, which prove unwieldy. A designer is consulted to decide which names should be retained and which should be changed. When a name is changed, the designer assigns the new name.

by a task. Three design-decision rules enable such situations to be identified and acted upon.

One rule reflects the asynchronous device-input/output task structuring criterion from CODARTS. [Gomaa93, pp. 190-191]

Rule: Asynchronous Device Interface

if
Transformation$_T$ is an Asynchronous Device Interface Object
then
if Transformation$_T$ is an Asynchronous Device Input Object
then create an asynchronous device-input Task$_{ADI}$
elseif Transformation$_T$ is an Asynchronous Device Output Object
then create an asynchronous device-output Task$_{ADI}$
else create an asynchronous device-io Task$_{ADI}$
fi
record the design decision and rationale in the design history for Task$_{ADI}$
denote the traceability between Transformation$_T$ and Task$_{ADI}$
fi

This rule recognizes each transformation in an input specification that inherits the concept Asynchronous Device Interface Object. Each such transformation forms the basis for a task, but the type of task created depends upon the leaf-level classification of the transformation. Tagging newly created tasks with a specific task type facilitates later decision-making and provides a readily understood reason for a task's existence in the design.

A similar rule corresponds to the periodic-device input/output task structuring criterion from CODARTS. [Gomaa93, pp. 191-193]

Rule: Periodic Device Interface

if
Transformation$_T$ is a Periodic Device Interface Object
then
if Transformation$_T$ is a Periodic Device Input Object
then create a periodic device-input Task$_{PDI}$
elseif Transformation$_T$ is a Periodic Device Output Object
then create a periodic device-output Task$_{PDI}$
else create a periodic device-io Task$_{PDI}$
fi
record the design decision and rationale in the design history for Task$_{PDI}$
denote the traceability between Transformation$_T$ and Task$_{PDI}$
fi

This rule recognizes each transformation in an input specification that inherits the concept Periodic Device Interface Object. As with the previous rule, the type of task created depends upon the leaf-level classification of the transformation. Segregating periodic device-interface tasks as input, output, and input/output enables functional cohesion to be considered along with temporal cohesion during the consideration of task mergers (see section 6.3, below).

CODARTS does not include task structuring criteria for interfaces to distributed, external subsystems, although Gomaa considers such issues in an extension to CODARTS intended for distributed applications. [Gomaa93, Chapter 25] The specification meta-model, described in Chapter 4 of this dissertation, extends COBRA by adding a terminator type to represent external subsystems and by adding an interface object type, Subsystem Interface Object, that provides an interface to external subsystems. This extension to COBRA allows large, real-time problems to be divided into smaller, more manageable, pieces that can then be modeled as a distributed system of subsystems,

communicating using loosely-coupled messages across a logical, or virtual, network.[2] To represent this model of distributed subsystems, one task is allocated for each external subsystem identified in the input specification. A designer can use each external subsystem to model a separate physical network, or as a placeholder in the design for each external subsystem that is being designed independently. The following rule identifies the need for an external subsystem input/output task.

Rule: External Subsystem

if

Transformation$_T$ is a Subsystem Interface Object

then

create an external subsystem-io Task$_{SI}$
record the design decision and rationale in the design history for Task$_{SI}$
denote the traceability between Transformation$_T$ and Task$_{SI}$

fi

Although not always needed, this explicit identification of interfaces to external subsystems can facilitate later integration of the subsystems into a larger design.

6.1.2 Rules For Identifying Internal Tasks

Several design approaches for real-time systems, including CODARTS, generate designs from a data/control flow diagram. By starting from a data/control flow diagram, the designer retains the option to create either a sequential design or a concurrent design. When generating a concurrent design from a data/control flow diagram, the structuring of interior transformations into concurrent tasks is considered a difficult challenge that

---

[2] Such logical networks might have actual counterparts in the various local area networks sometimes used for real-time applications, for example, ethernets, token bus broadband networks, or token-passing ring networks.

typically relies on loose heuristics that can only be applied well by experienced designers, using significant judgment. For example, Nielsen and Shumate, identify seven process selection rules that deal with internal transformations in a data/control flow diagram. [Nielsen88, Chapter 9] These process selection rules, based on an early version of DARTS [Gomaa84], are described too informally to serve as a basis for generating formal task-structuring rules. The challenge of structuring internal tasks can, however, be addressed more precisely by the approach proposed in this dissertation.

Assuming the existence of a semantically valid specification, design-decision rules can be defined to identify candidate tasks from among the internal transformations of a data/control flow diagram. The internal task-structuring criteria included in CODARTS provide a suitable starting point for defining the necessary rules. [Gomaa93, pp. 193-196]

One rule allocates a task for each Control Object found within a specification.

Rule: State-based Control

if
Transformation$_T$ is a Control Object
then
create a control Task$_C$
record the design decision and rationale in the design history for Task$_C$
denote the traceability between Transformation$_T$ and Task$_C$
fi

This rule corresponds to the Control Task structuring criterion from CODARTS. Since each Control Object encompasses a state-transition diagram and since a state-transition

diagram is sequential by definition, CODARTS uses the presence of a Control Object to justify the creation of a sequential thread of control.

A second rule allocates a task for each User-Role Interface Object in an input specification.

Rule: User Role

if
Transformation$_T$ is a User-Role Interface Object
then
create a user-role Task$_{UR}$
record the design decision and rationale in the design history for Task$_{UR}$
denote the traceability between Transformation$_T$ and Task$_{UR}$
fi

This rule reflects the CODARTS criterion for structuring tasks based on user roles. Each user of a system is assigned an interface task that performs sequential processing of user requests. Since modern operating systems allow a user to open and interact with multiple windows on a single terminal, each user-role task represents a separate logical user, rather than a physical user.

6.1.2.1 Internal Periodic Tasks

The CODARTS design method provides a criterion for identifying internal periodic tasks based upon finding internal transformations that need to execute periodically. [Gomaa93, pp. 193-194] An example cited by Gomaa identifies a transformation within a cruise control application that computes the distance traveled by an automobile. This computation can be made periodically in order to ensure that the cumulative distance traveled is fairly accurate at any given moment. This single CODARTS criterion can be

expanded into three rules. Each rule recognizes one type of transformation on a data/control flow diagram that can serve as the basis for an internal periodic task.

One rule reflects the periodic task criterion as described by Gomaa.

Rule: Periodic Algorithm

if
Transformation$_T$ is a Periodic Function
then
create a periodic internal Task$_{PI}$
record the design decision and rationale in the design history for Task$_{PI}$
denote the traceability between Transformation$_T$ and Task$_{PI}$
fi

Periodic tasks resulting from this rule simply execute periodically based on a timer.

The other two rules recognize transformations that execute periodically once activated by a Control Object. In some real-time applications, an algorithm might execute only when a situation arises where the algorithm is needed. Once the algorithm is activated it might then execute periodically. For example, consider an algorithm that tracks enemy targets to maintain a fix on their location so that weapons can be placed accurately. This algorithm, once activated, might periodically calculate a new position for the target. Activation of the algorithm might depend upon a decision that the target is hostile and should be attacked.

Two forms of controlled-periodic algorithm can be identified. In one form, an algorithm is activated by a Control Object and then executes periodically until it is deactivated by the Control Object.

Rule: Enabled Periodic Algorithm

if

Transformation$_T$ is an Enabled Periodic Function

then

create an enabled periodic Task$_{EP}$
record the design decision and rationale in the design history for Task$_{EP}$
denote the traceability between Transformation$_T$ and Task$_{EP}$

fi

For example, in a military air-defense application a target-tracking function might be activated when a weapons officer presses an engage button. The tracking function might then execute periodically until the target no longer appears, or until a weapons officer presses a disengage button.

In the second form of controlled-periodic algorithm, the algorithm is activated by a Control Object and then executes periodically until the algorithm reaches an internal decision that it is finished.

Rule: Triggered Periodic Algorithm

if

Transformation$_T$ is a Triggered Periodic Function

then

create a triggered periodic Task$_{TP}$
record the design decision and rationale in the design history for Task$_{TP}$
denote the traceability between Transformation$_T$ and Task$_{TP}$

fi

For example, imagine a videocassette system with a timed recording function that is given a stopping time and then periodically checks to see if it has reached either the

stopping time or the end-of-tape. Whenever either condition is detected the function deactivates itself.

6.1.2.2 Internal Asynchronous Tasks

The CODARTS design method provides a criterion for identifying internal asynchronous tasks based upon finding internal transformations that need to be executed on demand. [Gomaa93, p. 194] An example cited by Gomaa identifies a scheduler transformation within an elevator control application that receives requests for an elevator, finds an elevator to assign to the request, and then sends that elevator a schedule request. The scheduler transformation remains idle until an elevator request arrives and then completes the processing of that request before returning to an idle condition. This single CODARTS criterion can be expanded into three rules. Each rule recognizes one type of transformation on a data/control flow diagram that can serve as the basis for an internal asynchronous task.

One rule reflects the asynchronous task criterion as described by Gomaa.

Rule: Asynchronous Algorithm

if
Transformation$_T$ is an Asynchronous Function
then
create an asynchronous internal Task$_{AI}$
record the design decision and rationale in the design history for Task$_{AI}$
denote the traceability between Transformation$_T$ and Task$_{AI}$
fi

Asynchronous tasks resulting from this rule simply execute whenever work arrives.

The other two rules recognize transformations that execute on demand once activated by a Control Object. In one form, an algorithm is activated by a Control Object and then executes on demand until it is deactivated by the Control Object.

Rule: Enabled Asynchronous Algorithm

if
Transformation$_T$ is an Enabled Asynchronous Function
then
create an enabled asynchronous Task$_{EA}$
record the design decision and rationale in the design history for Task$_{EA}$
denote the traceability between Transformation$_T$ and Task$_{EA}$
fi

For example, in the elevator control application cited by Gomaa, an emergency detection feature might be added that can disable the scheduler transformation in the presence of unsafe conditions within the elevator system. When the emergency is cleared, the scheduler could be activated.

In the second form of controlled-asynchronous algorithm, the algorithm is activated by a Control Object and then executes on demand until the algorithm reaches an internal decision that it is finished.

Rule: Triggered Asynchronous Algorithm

if
Transformation$_T$ is a Triggered Asynchronous Function
then
create a triggered asynchronous Task$_{TA}$
record the design decision and rationale in the design history for Task$_{TA}$
denote the traceability between Transformation$_T$ and Task$_{TA}$
fi

For example, an on-line banking application might contain a transformation that processes customer transactions. The transformation could be triggered once the customer completes a controlled authentication process. Once triggered, the transformation might execute customer requests on demand until the customer indicates no more transactions are required.

6.2 Allocate Remaining Transformations to Tasks

Transformations in an input specification that do not lead directly to the creation of tasks include: Passive Device Interface Objects, Triggered Synchronous Functions, and Synchronous Functions. These remaining transformations must, however, be allocated to tasks within the evolving design. The CODARTS design method identifies criteria for allocating transformations to tasks based on control cohesion, sequential cohesion, and functional cohesion. Using these criteria, a number of rules can be specified to allocate the remaining transformations to tasks in the design. These transformation-allocation rules compose the second decision-making process required to structure tasks within the evolving design. These rules construct sequential chains through the unallocated transformations to reflect the thread of control for each task in the design.

6.2.1 Allocating Transformations Based on Control Cohesion

Reflecting a control cohesion criterion from the CODARTS design method, each Triggered Synchronous Function is allocated to the same task as the Control Object that

triggers the function. This allocation occurs because a Triggered Synchronous Function executes completely during the triggering state transition.

Rule: Triggered Synchronous Function

if

Transformation$_{TSF}$ is a Triggered Synchronous Function and
Transformation$_{CO}$ is a Control Object and
Transformation$_{TSF}$ receives a Signal or Trigger from Transformation$_{CO}$ and
Task$_{C}$ is derived from Transformation$_{CO}$

then

allocate Transformation$_{TSF}$ to Task$_{C}$
record the design decision and rationale in the design history for Task$_{C}$

fi

An example where this rule applies appears in a robot controller application described by Gomaa. [Gomaa93, Chapter 23] In the example, a Control Object, Control Robot, triggers six synchronous functions: Change Program, Start Program, End Program, Process Program Ended, Stop Program, and Resume Program. Based on a criterion for control cohesion, all of these transformations are allocated to the same task as Control Robot.

6.2.2 Allocating Transformations Based on Sequential Cohesion

By definition, transformations classified as Synchronous Functions or Passive Device Interface Objects must execute to completion once invoked. In many situations, this trait allows any such transformation to be allocated, based on sequential cohesion, to each task whose execution reaches the transformation. Four rules can be specified to recognize situations where sequential cohesion applies.

One rule allocates a Passive Device Interface Object to any task whose execution reaches that transformation.

Rule: Passive Device Interface Object

if

Transformation$_{PDIO}$ is a Passive Device Interface Object and
Transformation$_{PDIO}$ receives a Signal or Stimulus from
a Transformation$_{T}$ and
Transformation$_{T}$ traces to any Task$_{A}$

then

allocate Transformation$_{PDIO}$ to Task$_{A}$
record the design decision and rationale in the design history for Task$_{A}$

fi

An example were this rule applies can be found in the cruise control and monitoring application referred to in earlier sections of this dissertation. [Gomaa93, Chapter 22] In the example, a Passive Device Input Object, Gas Tank, is accessed by two transformations: Initialize MPG and Compute Average MPG. During the design process these two accessing transformations are allocated to two separate tasks. As a result, the Gas Tank transformation is allocated, based on sequential cohesion, to both tasks. The act of allocating transformations to tasks provides design information that can be used later (see Chapter 9) when information-hiding modules (see Chapter 8) are examined to determine which are shared among tasks and which are not. In general, when a transformation is allocated to multiple tasks and to one information-hiding module then the module is shared by the tasks.

Another rule allocates any Synchronous Function that sends a Response to the same task as the transformation that receives that Response. This rule models the sequential cohesion that exists between a subroutine and any caller of the subroutine.

Rule: Responding Synchronous Function

if

Transformation$_{SF}$ is a Synchronous Function and
Transformation$_{A}$ is any Solid Transformation and
Transformation$_{A}$ receives a Response from Transformation$_{SF}$ and
Task$_{T}$ is derived from Transformation$_{A}$

then

allocate Transformation$_{SF}$ to Task$_{T}$
record the design decision and rationale in the design history for Task$_{T}$

fi

An example where this rule applies appears in a robot controller system described by Gomaa. [Gomaa93, Chapter 23] In the example, a Synchronous Function, Process Sensor/Actuator Command, sends a Response, Sensor Value, to an Asynchronous Function, Interpret Program Statement. The Asynchronous Function forms the basis for an asynchronous task; the transformation, Process Sensor/Actuator Command, is allocated to the task based on sequential cohesion.

A third rule allocates any Synchronous Function that sends only locked-state events to a Control Object, but that sends no event or data flows to any other transformation, to the same task as the Control Object. This rule reflects the sequential cohesion between a Synchronous Function that generates locked-state events and the state-transition diagram that awaits those events.[3]

---

[3] Although this rule identifies a simple concept, a task calling a subprogram, the rule itself becomes fairly complex. This complexity results in part from the fact that the call **from** the task **to** the subprogram must be inferred from the data/control flow diagram in situations where a directed arc goes **from** the transformation representing the subprogram **to** the transformation representing the calling task. Additional complexity results from the fact that the transformation representing the subprogram must emit directed arcs only to the transformation representing the calling task. If these conditions are not satisfied, then the transformation representing the subprogram should instead be

Rule: Outputs Only Locked-State Events To A Control Object

if

$Transformation_{SF}$ is a Synchronous Function and
$Transformation_{SF}$ sends a Signal to $Transformation_{CO}$ and
$Transformation_{CO}$ is a Control Object and
$Transformation_{SF}$ sends no Signal to any other transformation and
$Transformation_{SF}$ sends no Stimulus and
$Task_{T}$ is derived from $Transformation_{CO}$ and
each $Signal_{S}$ from $Transformation_{SF}$ to $Transformation_{CO}$ is a
locked-state event and
$Transformation_{SF}$ and $Transformation_{CO}$ have identical cardinality

then

allocate $Transformation_{SF}$ to $Task_{T}$
record the design decision and rationale in the design history for $Task_{T}$

fi

An example where this rule applies can be found in an elevator control system described by Gomaa. [Gomaa93, Chapter 24] In the example, a transformation, Check This Floor, receives a data flow, Floor Number, consults a data store to determine if the elevator is to stop at that Floor Number, and, if appropriate, sends an event flow, Approaching Requested Floor, to a Control Object, Elevator Control. The event flow arrives only when the Control Object is waiting for it to arrive. If the transformation, Check This Floor, is classified as a Synchronous Function, then the rule, as specified above, applies.

A fourth rule allocates each Synchronous Function that meets certain output restrictions to the same task as the transformations that invoke that function. The output restrictions require that the Synchronous Function send any outputs in the form of Signals or Stimuli (see Chapter 4 for the definitions of these concepts) to either another

---

allocated, using another rule, to one or more other tasks in the evolving design.

Synchronous Function or a Passive Device Interface Object. Where these output restrictions are not satisfied, the Synchronous Function must be allocated to a task based application knowledge unavailable to the design-decision rules. The rule to allocate Synchronous Functions that meet output restrictions is specified below.

Rule: Synchronous Function With Synchronous Outputs

if

$Transformation_{SF}$ is a Synchronous Function and
$Transformation_{SF}$ sends no Signal or Stimulus to another $Transformation_{AT}$
unless $Transformation_{AT}$ is a Synchronous Function or a
Passive Device Interface Object and
$Transformation_{ST}$ is a Solid-Transform and
$Transformation_{ST}$ sends a Signal or Stimulus to $Transformation_{SF}$ and
$Transformation_{ST}$ is allocated to a $Task_T$

then

allocate $Transformation_{SF}$ to $Task_T$
record the design decision and rationale in the design history for $Task_T$

fi

An example where this rule might apply can be found in a cruise control and monitoring system, as described by Gomaa. [Gomaa93, Chapter 22] In the example, a transformation, Mileage Reset Buttons, sends two signals, MPG Reset and MPH Reset, one to each of two Synchronous Functions, Initialize MPG and Initialize MPH, respectively. Initialize MPG outputs only one Stimulus, Fuel Request, to a Gas Tank, while Initialize MPH outputs no Stimulus or Signal. Using the rule defined above, both of these Synchronous Functions should be allocated to the same task as the transformation Mileage Reset Buttons.

A fifth rule allocates each Synchronous Function meeting certain input restrictions to the same task as any transformation that invokes that function. The input restrictions require that the Synchronous Function receive any inputs in the form of Signals or Stimuli only from another Synchronous Function, a Triggered Synchronous Function or a Passive Device Interface Object. Where these input restrictions are not satisfied the Synchronous Function must be allocated to a task based on application knowledge unavailable to the design-decision rules.

Rule: Synchronous Function With Synchronous Inputs

if

Transformation$_{SF}$ is a Synchronous Function and
Transformation$_{ST}$ is a Synchronous Function or Triggered Synchronous Function or Passive Device Interface Object and
Transformation$_{ST}$ sends a Signal or Stimulus to Transformation$_{SF}$ and
Transformation$_{ST}$ is allocated to a Task$_{T}$ and
no other Transformation that is not a Synchronous Function and is not a Triggered Synchronous Function and is not a Passive Device Interface Object sends a Signal or Stimulus to Transformation$_{SF}$

then

allocate Transformation$_{SF}$ to Task$_{T}$
record the design decision and rationale in the design history for Task$_{T}$

fi

An example where this rule might apply can be found in a remote temperature sensor system, as described by Nielsen and Shumate. [Nielsen88, Appendix A] In the example, a Synchronous Function, Create ICP, receives a Stimulus, CP, from another Synchronous Function, Determine Message Type. Assuming that Determine Message Type is already allocated to a task in the design, then, using the rule defined above, Create ICP would be allocated to the same task as Determine Message Type.

6.2.3 Allocating Transformations Based on Functional Cohesion

Some cases that involve Synchronous Functions exhibiting sequential and functional cohesion cannot be decided solely from the data/control flow diagram. This type of situation arises when a Synchronous Function receives inputs from one or more transformations that are already allocated to a task and also sends outputs to one or more transformations that are already allocated to a task. Allocating the Synchronous Function to the most suitable task requires application knowledge that is unavailable to the design-decision rules. Such cases require the designer to select, where possible, the preferred allocation of a Synchronous Function to a specific task, or set of tasks. For example, consider a situation that arises in the Robot Controller case study discussed by Gomaa. [Gomaa93, Chapter 23] In the example, a Synchronous Function, Process Motion Command, receives a data flow, Motion Command, from an Asynchronous Function, Interpret Program Statement, and sends a data flow, Motion Block, to an Asynchronous Function. Each of the Asynchronous Functions is allocated to a separate task. The allocation of Process Motion Command to one or the other of the tasks requires application knowledge unavailable to the design-decision rules. In this case, the human designer can judge, based on functional cohesion, that the Process Motion Command should be allocated to the same task as Interpret Program Statement. The following rule is defined to refer such ambiguous situations to the designer and to elicit any guidance the designer might care to provide.

Rule: Designer Specifies Allocation (Last Preference)

if
Transformation$_{SF}$ is a Synchronous Function and
Transformation$_{A}$ is any Transformation and
Transformation$_{A}$ sends a Signal or Stimulus to Transformation$_{SF}$ and
Transformation$_{A}$ is allocated to a task
then
show the designer the Set$_{I}$ of transformations that send inputs to Transformation$_{SF}$ and the Set$_{O}$ of transformations that receive outputs from Transformation$_{SF}$ such that each member of Set$_{I}$ and Set$_{O}$ is already allocated to a task
ask the designer to select a transformation from the Set$_{I}$ or a transformation from Set$_{O}$ that should be allocated to the same task as Transformation$_{SF}$
if the designer cannot make a selection
then
allocate a Task$_{T}$ for Transformation$_{SF}$
allocate Transformation$_{SF}$ to Task$_{T}$
record the design decision and rationale in the design history for Task$_{T}$
else
if the designer selected a member from Set$_{I}$
then
denote the allocation of Transformation$_{SF}$ to the same task each as member of Set$_{I}$
else
denote the allocation Transformation$_{SF}$ and to the same task as the selected member of Set$_{O}$
fi
fi
fi

To better understand this rule, consider how the previous example is addressed. The rule recognizes that the Synchronous Function, Process Motion Command, cannot be allocated directly to a specific task. This recognition occurs because no other rule could make an allocation. The designer is shown the set of transformations from which Process

Motion Command receives inputs and the set of transformations to which Process Motion Command sends outputs. The designer is consulted, then, to indicate whether Process Motion Command should be allocated to the same task as one of the other transformations. If the designer cannot make an allocation, then Process Motion Command becomes the basis for an asynchronous task. If the designer makes an allocation, then the designer's selection is recorded for use in the following rule.

Rule: Implement Designer Allocation

if

$Transformation_{SF}$ is a Synchronous Function and
$Transformation_{A}$ is any transformation and
$Transformation_{A}$ is allocation to a $Task_{T}$ and
$Transformation_{SF}$ should be allocated to the same task as $Transformation_{A}$
(as indicated by the designer)

then

allocate $Transformation_{SF}$ to $Task_{T}$
record the design decision and rationale in the design history for $Task_{T}$

fi

This rule simply implements the designer's decision that a Synchronous Function should be allocated to the same task as a transformation that was allocated previously to a task.

6.3 Consider Task Mergers

Once a set of candidate tasks exists, consideration should be given to merging some of those candidate tasks based on cohesion criteria. The CODARTS design method identifies a number of task-cohesion criteria that might apply, including: sequential cohesion, temporal and functional cohesion, and task inversion. [Gomaa93, pp. 197-205] Sequential cohesion applies to tasks that cannot execute concurrently because of the

sequential nature of the interactions among them. Another form of sequential cohesion, which might be called mutual exclusion, allows candidate tasks to be combined when a control task, that is, a task formed based upon a Control Object, orders their execution so that they cannot execute together. Temporal cohesion applies to candidate tasks that execute with the same period, or with harmonic periods. Functional cohesion should be considered together with temporal cohesion because, in general, periodic tasks should not be combined when they represent functionally dissimilar processing; this is because the priorities assigned to functionally dissimilar tasks are likely to vary. Task inversion applies to candidate tasks that have cardinalities that exceed some reasonable threshold, and should, thus, be merged, where required to optimize the design or to reduce task overhead.

Defining these task-cohesion criteria as a set of design-decision rules presents several challenges. First, multiple cohesion criteria might apply in a given situation. This means that either a designer must be consulted to select from among the various options, or that a preferred ordering must be built into the rules so that an appropriate choice is made from among competing decisions. A second challenge occurs because some of the decision criteria might be applied in some circumstances, but not in others, based on knowledge of the target environment or on a designer's judgment. Moreover, inexperienced designers will probably not be in any position to make the required judgments.

To meet these challenges, two strategies are adopted when defining design-decision rules for considering task mergers. The first strategy establishes a preferred ordering among competing decisions. Seven levels of preference are used. The preference ordering for each rule is specified when the rule is defined; however, an explanation of the rationale behind assigning these preferences is deferred until after all of the rules are explained. The second strategy ensures that certain design-decision rules, rules requiring judgments unsuitable for an inexperienced designer, will not be used unless an experienced designer is at work.

6.3.1 Rules for Combining Tasks Based on Mutual Exclusion

Mutual exclusion provides the basis for two rules for combining tasks. One rule recognizes cases where a control task manages a set of tasks in such a manner that their execution is mutually exclusive.

Rule: Controlled Task Exclusive Execution (First Preference)

if
  $Task_{E1}$ is an enabled asynchronous or enabled periodic task and
  $Task_{E2}$ is an enabled asynchronous or enabled periodic task and
  for every $Transformation_T$ from which $Task_{E1}$ or $Task_{E2}$ is derived
    $Transformation_T$ is a member of **Exclusion Group**$_{EG}$
then
  if either $Task_{E1}$ or $Task_{E2}$ is an enabled periodic task
  then combine $Task_{E1}$ and $Task_{E2}$ into a single enabled periodic $Task_{ME}$
  else combine $Task_{E1}$ and $Task_{E2}$ into a single enabled asynchronous $Task_{ME}$
  fi
  record the design decision and rationale in the design history for $Task_{ME}$
fi

An example of where this rule applies can be found in a cruise control and monitoring system application described by Gomaa. [Gomaa93, Chapter 22] In this example, a Control Object, Cruise Control, manages the operation of three Enabled Periodic Functions: Maintain Speed, Resume Cruising, and Increase Speed. Each of these functions adjusts a throttle in order to affect the speed of an automobile. The state-transition diagram within the Control Object enables and disables the three functions in such a manner that their execution is mutually exclusive. This relationship between the Control Object and any mutually exclusive, enabled functions is determined through a manual examination of the state-transition diagram. For each such relationship discovered, an **Exclusion Group** is created to link the Control Object with the mutually-exclusive, enabled functions that it controls.

A second rule recognizes situations where mutual exclusion exists among sets of state-independent tasks, including both asynchronous and periodic tasks, due to application-specific restrictions.

Rule: Independent Task Exclusion Execution (Fourth Preference)

if

$Task_{I1}$ is a asynchronous internal or periodic internal or combined internal task and
$Task_{I2}$ is a asynchronous internal or periodic internal task and
for every $Transformation_T$ from which $Task_{I1}$ or $Task_{I2}$ is derived
$Transformation_T$ is a member of **Exclusion Group**$_{EG}$

then

combine $Task_{I1}$ and $Task_{I2}$ into a combined internal $Task_{CI}$
record the design decision and rationale in the design history for $Task_{CI}$

fi

A remote temperature sensor application, described by Nielsen and Shumate, provides an example where this rule applies. [Nielsen88, Appendix A] The data/control flow diagram for the application includes an Asynchronous Function, Get New DP, and a Periodic Function, Send Old DP. Careful reading of the accompanying textual specification reveals that these two transformations implement a stop-and-wait protocol. Only one of the two transformations can be active at any time. Knowing this, a designer places these two transformations into the same exclusion group, and then the rule specified above considers the transformations as candidates for merger, based on mutual exclusion.

6.3.2 Rules for Combining Tasks Based on Sequential Cohesion

A form of sequential cohesion occurs in situations where a control task is locked in some state awaiting an event from another task. Since the control task cannot continue until the awaited event arrives, the execution of the control task and the task it controls must be sequential. A rule to recognize this situation is given below.

Rule: State-Dependent Input To Control Task (Second Preference)

if $Task_C$ is a control task derived from a $Transformation_{CO}$ and
$Transformation_{CO}$ is a Control Object and
$Task_A$ is any task derived from a $Transformation_T$ and
($Transformation_T$ is a State-Dependent Function or an Asynchronous Function) and
$Transformation_T$ sends one or more Signals to $Transformation_{CO}$ and
for each $Signal_S$ from $Transformation_T$ to $Transformation_{CO}$
$Signal_S$ is a **locked-state event** for $Transformation_{CO}$ and
$Task_C$ and $Task_A$ have identical cardinality

then
combine $Task_C$ and $Task_A$ into a single control $Task_{CA}$
record the design decision and rationale in the design history for $Task_{CA}$

fi

This rule represents a complex situation identified in the CODARTS criteria for control cohesion, see particularly the third case discussed by Gomaa. [Gomaa93, pp. 202-203] The concept **locked-state event**, referred to in the rule, warrants further elaboration.

Each Control Object in an input specification encompasses a state-transition diagram, or STD. A state in a STD might be called a **non-locked state** when the arrival of any one of a number of events from several sources causes a transition from that state. A state in a STD might be called a **locked state** when transition out of that state requires the arrival of a single, specific event (or any one of a set of events from a single, specific transformation). Any event that causes a STD to transition from a **locked state** can be called a **locked-state event**. The **locked-state events** for each STD are established through a manual review of the STD and its related data/control flow diagram. The cardinality of the two tasks to be combined must be identical because otherwise each **locked-state event** recognized by the rule would actually represent multiple events each coming from a different source (that is, multiple instances of the source transformation), rather than as a separate instance of the same event type exchanged between multiple instances of a source task and a destination task. The STD state receiving these multiple events fails the definition of a **locked state**.

Another rule derived from the CODARTS criteria for sequential cohesion recognizes, more simply, situations where a periodic task provides exclusive input to a control task. In such cases, the two tasks can be combined because the control task will not execute until after an input arrives from the periodic task. These two tasks, then, can

execute sequentially. The specification of a rule to recognize this situation is given below.

Rule: Exclusive Input To Control Task (Third Preference)

if
$Task_C$ is a control task and
$Task_P$ is a periodic device-input task or a periodic device-io task or
a periodic internal task and
$Task_C$ and $Task_P$ have identical cardinality and
$Task_C$ receives inputs only from $Task_P$ and
$Task_P$ sends outputs only to $Task_C$ and
$Task_C$ is not derived from a State-Dependent Function
then
combine $Task_C$ and $Task_P$ into a single control $Task_{CP}$
record the design decision and rationale in the design history for $Task_{CP}$
fi

The cruise control application described by Gomaa provides an illustration of where this rule can apply. [Gomaa93, Chapter 22] In the example, a Control Object, Calibration Control, receives two input events from a Periodic Device Input Object, Calibration Buttons. Assume that Calibration Control had been allocated previously to a control task and that Calibration Buttons had been allocated to a periodic device-input task. Also assume that the control task had not already been combined with a state-dependent task. Since Calibration Control receives no other inputs, and since Calibration Buttons sends outputs to no other task, these two tasks can be combined because the periodic device-input task provides exclusive input to the control task.

6.3.3 Rules for Combining Tasks Based on Temporal Cohesion

Temporal cohesion allows periodic tasks with identical periods to be combined. Temporal cohesion also permits periodic tasks with unequal periods to be combined

when the periods of the tasks meet three conditions: 1) the periods are multiples of one another, 2) the periods are within an order of magnitude,[4] and 3) the periods are closer together than any other pair of periods that meet the first two criteria[5].

According to CODARTS, combining tasks with unequal periods requires judgment on the part of a designer; [Gomaa93, p. 199] thus, when situations of this type occur they should be referred to an experienced designer for a decision. When combining periodic tasks, care should be taken to ensure that only tasks of similar priority are merged. At this stage in the design process, when priorities are not yet assigned, two factors are considered to judge the relative importance of tasks. First, the type of task must be identical. This allows periodic device-input tasks, periodic internal tasks, periodic device-output tasks, and periodic device-input/output tasks to be combined separately from each other in case the priorities of these functions differ. Second, for tasks with unequal periods, the periods must be within an order of magnitude before the tasks are eligible to be combined. Even in this case a designer is consulted to make the final determination.

The CODARTS temporal cohesion criteria lead to three design-decision rules. One rule combines a periodic task with multiple instances into a single periodic task (in CODARTS, this is called task inversion). This makes sense because a multiple-instance

---

[4] This condition exists because tasks with widely separated, but harmonic, periods are likely to be assigned different priorities during design configuration. Combining such tasks would prevent the assignment of differing priorities.

[5] This condition exists so that, when multiple pairs of harmonic tasks are candidates to be combined, pairs of harmonic tasks with the closest periods, and thus likely the most similar priorities, are considered before pairs of harmonic tasks whose periods are farther apart, and thus likely to have less similar priorities.

task is certain to be functionally cohesive and each instance of the task will have an identical period. Once a periodic task with multiple instances is inverted, the resulting task then becomes eligible to be combined with other, single-instance, periodic tasks of the same functional type. A rule to recognize and invert periodic tasks with multiple instances is specified below.

Rule: Periodic Task With Multiple Instances (Fifth Preference)

if

$Task_P$ is a periodic device-input task or a periodic device-output task or a periodic device-io task or a periodic internal task and
$Task_P$ has a cardinality greater than one

then

invert $Task_P$ into a single task
record the design decision and rationale in the design history for $Task_P$

if

An example application of this rule might be found in an elevator control system. Suppose that on each of three floors in a building two elevator buttons exist: pressing one requests an upwards ride on an elevator and pressing the other requests a downwards ride. If these buttons are modeled via a Periodic Device Input Object with a cardinality of six, then a periodic device-input task with a cardinality of six will be allocated to poll these buttons. This six-instance task can be converted into a single task because each instance of the task performs the same function with the same periodicity.

Another rule, specified below, recognizes pairs of single-instance periodic tasks of the same type and combines them into a single task.

Rule: Periodic Tasks With Identical Periods (Fifth Preference)

if

$Task_{P1}$ is a periodic device-input task or a periodic device-output task or
a periodic device-io task or a periodic internal task and
$Task_{P2}$ is of identical type to $Task_{P1}$ and
both $Task_{P1}$ and $Task_{P2}$ have a cardinality of one and
$Task_{P1}$ and $Task_{P2}$ have identical periods

then

combine $Task_{P1}$ and $Task_{P2}$ into a single $Task_{P}$
record the design decision and rationale in the design history for $Task_{P}$

fi

A situation where this rule applies is depicted in a cruise control and monitoring system example described by Gomaa. [Gomaa93, Chapter 22] In the example, two periodic device-input tasks are created: one, based on a Periodic Device Input Object, Brake, monitors a brake sensor, while the other, based on a Periodic Device Input Object, Engine, monitors an engine sensor. The period of each task is given as 100 milliseconds. Since both tasks are periodic device-input tasks with a single instance and since both have the same period, the tasks can be combined. Should the application call for a third sensor, perhaps an air bag sensor, that is also polled every 100 milliseconds, then repeated application of the rule would combine the task monitoring the air bag sensor with the task that monitors the brake and engine sensors.

The final rule defined to reflect the temporal cohesion criteria from CODARTS recognizes pairs of periodic tasks that might be combined even though their periods are not identical. Such a combination should be considered when a pair of periodic tasks of the same type have unequal periods that are multiples of one another, provided that the periods are within an order of magnitude. The periods must be multiples of one another

so that a combined task might perform some functions during each periodic invocation, while reserving other functions for execution only during certain periodic invocations. The periods of the two tasks should be within an order of magnitude so as to improve the probability that the processing performed by each of the tasks has a similar priority. Among all the pairs of tasks that satisfy these conditions, first consideration should be given to combining the pair of tasks with the closest periods. Again, this improves the probability that the tasks have a similar priority. Tasks without identical periods should not be combined without consulting a designer because the priorities of the tasks might not be similar enough to warrant combining the tasks, even though all the other criteria are satisfied. Since a designer must be consulted, this rule should only be used when an experienced designer is available to make the final decision. The rule is specified below.

Rule: Periodic Tasks With Harmonic Periods (Sixth Preference)

if
designer is experienced and
$Task_{P1}$ is a periodic device-output task or a periodic internal task or
a periodic device-input task or a periodic device-io task and
$Task_{P2}$ is of identical type to $Task_{P1}$ and
$Task_{P1}$ and $Task_{P2}$ have unequal periods and
$Task_{P1}$ and $Task_{P2}$ have periods that are multiples of one another and
$Task_{P1}$ and $Task_{P2}$ have periods within an order of magnitude and
$Task_{P1}$ and $Task_{P2}$ have the closest periods that satisfy the preceding two criteria and
both $Task_{P1}$ and $Task_{P2}$ have a cardinality of one
then
ask the designer whether these tasks should be combined
if the designer indicates the tasks should be combined
then combine $Task_{P1}$ and $Task_{P2}$ into a single $Task_{P}$
record the design decision and rationale in the history for $Task_{P}$
fi
fi

An example were this rule applies can be found in the cruise control and monitoring system described by Gomaa. [Gomaa93, Chapter 22] As the concurrent design for the example application progresses, three periodic internal tasks are created: Determine Speed and Distance, Compute Average Mileage, and Check Maintenance Need. Assume that these tasks execute with periods of 100 milliseconds, one second, and two seconds, respectively. The periods of each pair of these task prove to be multiples. The first task should not be combined with either of the others because its period is not within an order of magnitude. The second pair of tasks, Check Maintenance Need and Compute Average Mileage, might be combined because they are within an order of magnitude and there exists no other pair of periodic internal tasks to consider. This possible combination is referred to an experienced designer, if one is available. If no experienced designer is available, the tasks are not combined.

6.3.4 Rules for Combining Tasks Based on Task Inversion

Some run-time systems impose a large overhead for switching between tasks. Such overhead can become a significant concern in real-time applications. To enable an optimization of the design, the task inversion criterion identified in CODARTS permits multiple instances of a task to be replaced by a single task that performs the required context switching inside the task. Taking such a step requires detailed design of a context switching mechanism internal to the inverted tasks, thus requiring extra work and diminishing the understandability of the design. For these reasons, inversion of tasks

should be performed only where absolutely necessary to meet performance objectives. A rule to reflect this criterion is specified below.

Rule: Task Inversion (Last Preference)

if

$Task_I$ has a cardinality at or beyond the task-inversion threshold and the task-inversion threshold exceeds one

then

invert $Task_I$ into a single task

record the design decision and rationale in the design history for $Task_I$

fi

This rule depends upon a task-inversion threshold that is defined as part of a target environment description used for a particular design-generation session. Whenever a task has a number of instances that exceed the task-inversion threshold then the multiple-instance task is inverted into a single task.

An example that might call for task inversion can be illustrated with a data communications application, implemented in a UNIX environment. Assume that a task is created to manage each data connection opened with another system. As the number of connections increases over time, the number of UNIX processes increases. Process switching within a UNIX target environment can become quite time-consuming. In this case, as an optimization step, the task-inversion rule allows the connection management tasks to be replaced by a single task that switches among the connections. During detailed design, the mechanism for switching context between connections must be defined inside the inverted task.

6.3.5 Rationale for Preferred Rule Orderings

The process that considers task mergers uses a preferred ordering among the rules to reflect preferences for certain forms of cohesion over other forms. In general, preference is given to stronger forms of cohesion over weaker forms. Among the forms of cohesion relevant to this discussion, functional cohesion is strongest, followed in weakening order by sequential cohesion and temporal cohesion. Determination of functional cohesion relies, for the most part, on application-specific knowledge unavailable to the design-decision rules. For this reason, functional cohesion is not considered when assigning preferred orderings. Instead, first preference falls to sequential cohesion, and temporal cohesion receives next preference. The task-inversion criterion is assigned **Last Preference** because task inversion can be viewed as an optional optimization step. Since multiple rules exist for sequential cohesion and for temporal cohesion, consideration must be given to distinguishing among the rules within each category based upon some rational differences.

Among the rules for sequential cohesion, which include mutual exclusion, preferences must be distinguished because a control task should not be combined with both a periodic task that provides exclusive input to the control task and another task for which the control task waits in a locked state. Combining such tasks may lead to a situation where the periodic function cannot be performed in time because the other, independent function has yet to relinquish control. When a control task qualifies for combination based on both of these rules for sequential cohesion, weak preference is

given to merging a control task with a task that it controls or with an aperiodic task. This preference is given because the periodic task in question might qualify for combination with other periodic tasks, based on temporal cohesion. Another situation that must be considered occurs when a state-dependent task qualifies for combination with its control task, based on sequential cohesion, and with another state-dependent task, based on mutual exclusion. In this situation a weak preference is given to combining tasks based on mutual exclusion. This weak preference results from a recognition that after mutually exclusive tasks are combined then: 1) the task that controls the combined task might still qualify for combination with a periodic or aperiodic task, based on sequential cohesion, or 2) the combined task might still qualify for combination with its control task, based on sequential cohesion. This analysis suggests that, among the rules for mergers involving a control task, the following preferences should be assigned to the rules that are derived from criteria for mutual exclusion and sequential cohesion.

- **First Preference** is given to combining state-dependent tasks based on mutual exclusion.
- **Second Preference** is given to combining tasks based on sequential cohesion between control tasks and state-dependent tasks.
- **Third Preference** is given to combining tasks based on sequential cohesion between control tasks and periodic tasks.

The remaining rule addressing sequential cohesion, in the form of mutual exclusion among state-independent tasks, simply receives the next available preference, or **Fourth Preference**.

Among the rules for temporal cohesion preference is given to combining periodic tasks with identical periods over combining periodic tasks with unequal periods. This reflects a judgment that tasks with identical periods exhibit a stronger form of temporal cohesion than tasks with unequal, but harmonic, periods. As a result of this distinction between two forms of temporal cohesion, rules for combining periodic tasks with identical periods are given **Fifth Preference** overall. This leaves the rule for combining tasks with harmonic periods as the **Sixth Preference**.

Once the consideration of task mergers is completed, the basic, application-oriented task structure for the concurrent design exists. Additional tasks might be added, during the next decision-making process, to monitor resources or, during a later decision-making process, to implement message queues.

6.4 Consider Resource Monitors

Once the application-oriented tasks in a design are established, the evolving design can be further examined to identify cases where resource-monitor tasks are needed. Currently, two design-decision rules are specified to allocate resource-monitor tasks whenever multiple tasks share access to a passive output device or a passive input/output device. These rules reflect the CODARTS criterion, Resource Monitor Task. [Gomaa93, p. 193] The first rule recognizes cases where multiple tasks access the same

passive device. Since these cases are more general than those recognized by the next rule, this rule receives first preference.

Rule: Multi-Task Access (First Preference)

if
Transformation$_{PDO}$ is a Passive Device Output Object or a
Passive Device IO Object and
Transformation$_{PDO}$ is accessed by a Task$_{AT1}$ and
Transformation$_{PDO}$ is accessed by a Task$_{AT2}$
then
create a resource-monitor Task$_{RM}$
for each Task$_{I}$ that accesses Transformation$_{PDO}$
deallocate Transformation$_{PDO}$ from a Task$_{I}$
record the design decision and rationale in the design history for Task$_{I}$
rof
allocate Transformation$_{PDO}$ to Task$_{RM}$
record the design decision and rationale in the design history for Task$_{RM}$
fi

This rule might apply in any real-time application where multiple tasks log information to a single printer. Such applications require a resource-monitor task to ensure data is logged in the order received.

Another rule recognizes cases where a multiple-instance task accesses a passive output or input/output device. This rule, specified below, must take into account whether the accessed device resides in an Aggregation Group. If so, then the device is not shared but, rather, must be replicated within each instance of the accessing task.

Rule: Multi-Instance Task Access (Second Preference)

if

Transformation$_{PDO}$ is a Passive Device Output Object or a
Passive Device IO Object and
Transformation$_{PDO}$ receives an Input from Device$_{PD}$ or
sends an Output to Device$_{PD}$ and
Device$_{PD}$ is not in an **Aggregation Group** and
Task$_{AT}$ accesses Transformation$_{PDO}$ and
Task$_{AT}$ has a cardinality exceeding one

then

create a resource-monitor Task$_{RM}$
deallocate Transformation$_{PDO}$ from Task$_{AT}$
record the design decision and rationale in the design history for Task$_{AT}$
allocate Transformation$_{PDO}$ to Task$_{RM}$
record the design decision and rational in the design history for Task$_{RM}$

fi

An example where this rule applies appears in the elevator control system described by Gomaa. [Gomaa93, Chapter 24] In the example, a number of passive output devices appear, including three types of lamps: Floor Lamps, Direction Lamps, and Elevator Lamps. Each of these devices receives data from transformations that are grouped into a multiple-instance task, Elevator Controller. Two of the devices, Floor Lamps and Direction Lamps, do not reside in an Aggregation Group. The rule specified above allocates one resource-monitor task for each of this pair of devices. The third device, Elevator Lamps, is a member of an Aggregation Group. The rule specified above does not allocate a resource-monitor for the Elevator Lamps.

6.5 Review Task Structure and Consider Renaming Tasks

After task structuring is complete, the designer is then given an opportunity to review the task structure. The review can be used to examine the results obtained by

applying Task Structuring Knowledge to the input data/control flow diagram. If the designer is dissatisfied with the results, then they can be discarded. A single rule, not given here, drives the task review and renaming that completes the structuring of tasks.